NUMA 向けのアプリケーションの最適化

この記事は、インテル® ソフトウェア・ネットワークに掲載されている「Optimizing Applications for NUMA」の日本語参考訳です。

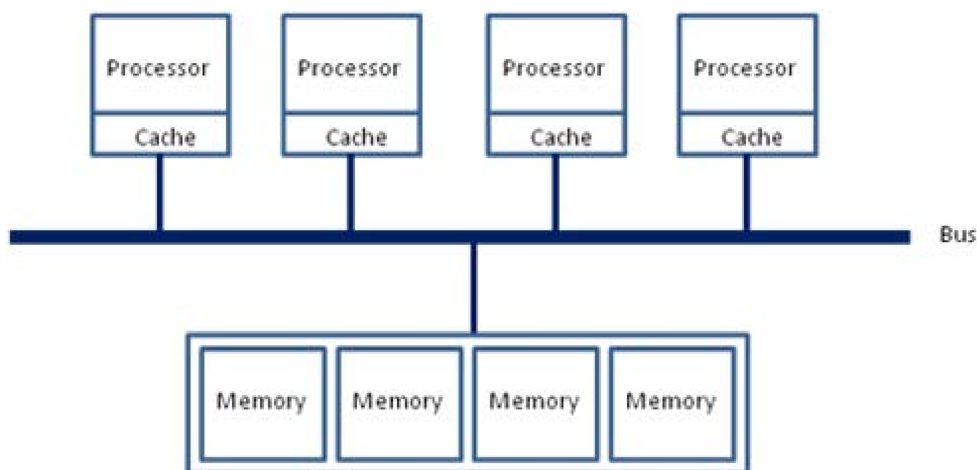
概要

NUMA (Non-uniform Memory Access) とは、共有メモリー型アーキテクチャーで、マルチプロセッサー・システムにおけるプロセッサーのメインメモリーの配置を表します。ほかの多くのプロセッサー・アーキテクチャーの特徴と同様に、NUMA を知らなければ、アプリケーションのメモリー・パフォーマンスを最適化することはできません。幸いなことに、NUMA ベースのアプリケーションのパフォーマンスの問題を軽減したり、並列アプリケーションに NUMA アーキテクチャーを役立てる方法は知られています。例えば、プロセッサー・アフィニティー、暗黙のオペレーティング・システム・ポリシーを使用するメモリー割り当て、明示的なディレクティブによりシステム API を使用するメモリーページの割り当てと移動などがあります。

この記事は、「マルチスレッド・アプリケーションの開発のためのガイド」の一部で、インテル® プラットフォーム向けにマルチスレッド・アプリケーションを効率的に開発するための手法について説明します。

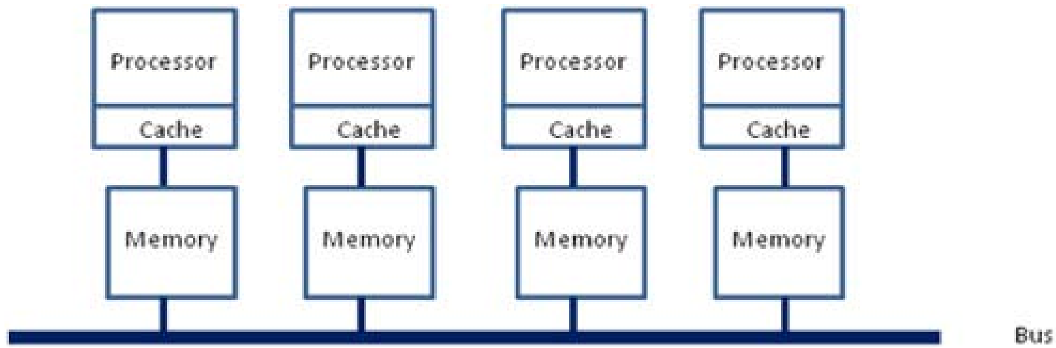
背景

NUMA を理解する最良の方法は、従来の UMA (Unified Memory Access) と比較することでしょう。UMA メモリー・アーキテクチャーでは、次の図に示すように、すべてのプロセッサーが 1 つのバス (あるいはほかのインターコネクト) を介して共有メモリーにアクセスします。



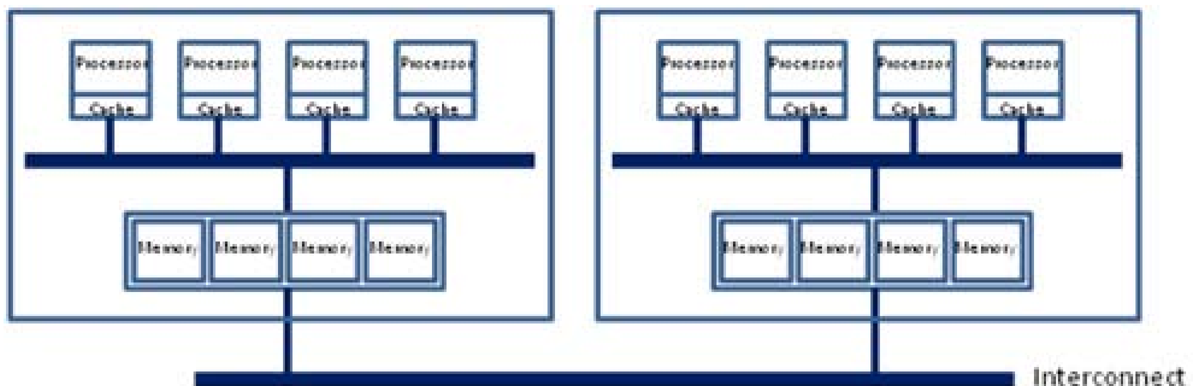
各プロセッサーが同じ共有バスを使用してメモリーにアクセスするため、すべてのプロセッサーのメモリーアクセス時間が均一になることから UMA (Unified Memory Access) という名前が付けられました。アクセス時間は、メモリー内のデータの位置に依存しないことに注意してください。つまり、取得するデータがどの共有メモリーモジュールに格納されていても、アクセス時間は同じになります。

NUMA 共有メモリー・アーキテクチャーでは、各プロセッサーが個別のローカルメモリーを保持し、それらに直接アクセスするため、パフォーマンスの面において大きな利点があります。さらに、次の図に示すような共有バス (あるいはその他のインターコネクト) を使用して、別のプロセッサー (リモート) のメモリーモジュールにアクセスすることもできます。



アクセスするデータの位置によりメモリーアクセス時間が均一でないことから NUMA (Non-uniform Memory Access) という名前が付けられました。データがローカルメモリーにある場合、アクセス時間は速くなります。データがリモートメモリーにある場合、アクセス時間は遅くなります。階層的な共有メモリー構成である NUMA アーキテクチャーの利点は、高速アクセス可能なローカルメモリーにより、一般的なケースにおいてアクセス時間を短縮できることです。

現代のマルチプロセッサー・システムでは、次の図に示すように、これらの基本アーキテクチャーを組み合わせています。



この複雑な階層構成では、プロセッサーは単一のまたはマルチコア CPU パッケージ、あるいはノード上の物理的な位置によってグループ化されています。ノード内のプロセッサーは、UMA 共有メモリー・アーキテクチャーに従ってメモリーモジュールへのアクセスを共有します。同時に、共有インターコネクトを使用してリモートノードにアクセスすることも可能ですが、NUMA 共有メモリー・アーキテクチャーによりパフォーマンスは低下します。

アドバイス

NUMA 共有メモリー・アーキテクチャーにおいてパフォーマンスを管理する上で重要なことは、**プロセッサー・アフィニティー**と**データの配置**の 2 つです。

プロセッサー・アフィニティー

Linux* や Windows* などを始めとする近年の汎用オペレーティング・システムは、スケジューラーを使用してアプリケーション・スレッドをプロセッサー・コアに割り当てます。スケジューラーは、システムの状態とさまざまなポリシー (例えば、「コア間で負荷が均一になるようにする」や「いくつかのコアにスレッドを集中させ、残りのコアはスリープ状態にする」など) の元に、アプリケーション・スレッドを物理コアに割り当てます。あるスレッドは、別のスレッドから実行の機会を得て、そのスレッドと交代し待機状態になるまで、割り当てられたコアで一定時間実行

されます。別のコアが利用可能になると、スケジューラーはタイムリーな実行とポリシーを達成するため、待機中のスレッドを移動することがあります。

1 つのコアから別のコアへスレッドを移動すると、NUMA 共有メモリー・アーキテクチャーでは、スレッドとそのローカルメモリー割り当ての関係が変わり問題が生じます。例えば、ノード 1 内のコアで実行を開始したスレッドがノード 1 でメモリーを割り当てた場合、そのスレッドがノード 2 内のコアに移動されると、以前に格納したデータはリモートとなり、メモリーアクセス時間が非常に遅くなります。

プロセッサー・アフィニティーについて考えてみましょう。プロセッサー・アフィニティーとは、ほかに利用可能なプロセッサー・コアがある場合でも、特定のプロセッサー・コアへのスレッド/プロセスの割り当てを保持し続けることです。システム API を使用したり、OS データ構造体 (アフィニティー・マスクなど) を変更することで、特定のコアまたはコアのセットをアプリケーション・スレッドに関連付けることができます。そして、スケジューラーがそのスレッドの存続期間におけるスケジュールを決定する際に、このアフィニティーが順守されます。例えば、クアッドコア CPU パッケージ 0 上にあるコア 0 からコア 3 のいずれかでのみ実行するようにスレッドを設定することができます。この場合、スケジューラーがコア 0 からコア 3 の中から選択すると、別のパッケージへスレッドを移動することはしません。

プロセッサー・アフィニティーを利用することで、スレッドが使用するメモリーをローカルに保つことができます。ただし、いくつかの影響も考慮すべきです。一般に、プロセッサー・アフィニティーは、スケジューラーの選択肢を制限し、本来であればより適切なリソース管理が可能な場合でもリソースの競合を招き、システムのパフォーマンスを大幅に低下させる可能性があります。スケジューラーが待機中のスレッドをアイドル中のコアに割り当てるのを妨げ、さらに別のノードで実行してもメモリーアクセス時間の遅延を十分補える場合、アプリケーションのパフォーマンスに影響を与えます。

プログラマーは、特定のアプリケーションと共有システムにおいてプロセッサー・アフィニティーを利用すべきかどうか注意深く検討する必要があります。一部のシステムで提供されているプロセッサー・アフィニティー API は、明示的なディレクティブに加えて、スケジューラーへの優先度の「ヒント」やアフィニティーの「提案」をサポートしています。スレッドの割り当てでは、明確に構造を強制するよりも、これらの「ヒント」や「提案」を使用するほうが、一般に最適なパフォーマンスが得られ、リソースの競合が発生しやすい場合にはスケジューリングの選択肢を制限しなくて済みます。

暗黙のメモリー割り当てポリシーによるデータの配置

多くのオペレーティング・システムが NUMA に適したデータの配置を透過的にサポートしています。シングルスレッド・アプリケーションがメモリーの割り当てを行う場合、プロセッサーは単に要求スレッドのノード (CPU パッケージ) に関連付けられている物理メモリーのメモリーページを割り当てることで、スレッドに対してローカルでアクセス・パフォーマンスが最適になるようにします。

オペレーティング・システムによっては、割り当て要求があっても最初のメモリーアクセスまで、メモリーページの割り当てを行わないものもあります。なぜそのように振る舞うのか、この利点を理解するために、起動時にメインの制御スレッドでメモリーの割り当てを行い、さまざまなワーカースレッドを生成し、アプリケーションの処理とサービスに長時間を費やすマルチスレッド・アプリケーションについて考えてみましょう。メモリー割り当てを要求するスレッドに対してローカルなメモリーページを割り当てることは合理的に思われるかもしれませんが、実際にはデータにアクセスするワーカースレッドに対してローカルなメモリーページを割り当てるほうがより効果的です。そうすることで、オペレーティング·システムは最初のアクセス要求を受け取った時に、要求元のノードの位置に応じてコミットページを割り当てることができます。

この 2 つのポリシー (「最初のアクセスに対してローカル」と「最初の要求に対してローカル」) は、アプリケーション・プログラマーがプログラムを配置する上で NUMA を理解していることがいかに重要であるかを示しています。ページ割り当てポリシーが最初のアクセスに基づいている場合、プログラマーは起動時にデータアクセス順序を

注意深く設計することで、オペレーティング・システムに最適なメモリー割り当ての「ヒント」を与えることができます。ページ割り当てポリシーが要求元の場所に基づいている場合、プログラマーはプロビジョニング・エージェントとして設計された初期化スレッドや制御スレッドではなく、後にデータをアクセスするスレッドによってメモリー割り当てが行われるようにすべきです。

複数のスレッドが同じデータにアクセスする場合、スレッドを同じノードに配置するのが最適です。そうすることで、ノード上のローカルメモリーを 1 回割り当てるだけで、すべてのスレッドがそのデータを使用できるようになります。これは、例えば、実際にデータが必要になる前にデータを要求することでパフォーマンスを向上させるプリフェッチに使用できます。その場合、NUMA アーキテクチャーではパフォーマンス・スピードアップの特性を活かすために、実際にデータを使用するスレッドに対してローカルな場所にデータを配置する必要があります。

オペレーティング・システムにより 1 つのノードの物理メモリーリソースが完全に消費されている場合、同じノードのスレッドからメモリー要求があると、通常はリモートノードにある第 2 候補の場所が割り当てられます。メモリーを大量に消費するアプリケーションでは、個々のスレッドに必要なメモリーサイズを正確に求め、アクセスするスレッドに対してローカルな場所にデータを配置します。

多数のノードに分散したスレッドが同じデータプールをランダムにアクセスする場合、データがすべてのノードにわたって均一に分散されるようにします。そうすることで、メモリーアクセスの負荷が分散され、1 つのノードにアクセスが集中するのを回避できます。

明示的なメモリー割り当てディレクティブによるデータの配置

NUMA ベースのシステムでデータを配置する別の方法として、メモリーページの割り当て場所を明示的に指定するシステム API の使用が挙げられます。そのような API の一例として、Linux* 向けの libnuma ライブラリーがあります。

この API を使用して、プログラマーは仮想メモリーアドレス範囲を特定のノードに関連付けたり、メモリー割り当てシステムコールでノードを指定することができます。この機能を使用することで、プログラマーは割り当てスレッドや最初にアクセスするスレッドに関係なく、特定のデータセットを配置することができます。これは、複雑なアプリケーションでワーカースレッドの代役としてメモリー管理スレッドを使用する場合に便利です。あるいは、存続期間が短く、データ要件が予測可能なスレッドを多数生成するアプリケーションでも役立ちます。このような制御は、プリフェッチにおいても大きな利点があります。

この方法のマイナス面は、いうまでもなく、プログラマーがメモリー割り当てとデータ配置を管理しなければならないことです。データが適切に配置されないと、デフォルトのシステム動作よりもパフォーマンスが大幅に低下することがあります。また、明示的なメモリー管理は、アプリケーション全体にわたってプロセッサー・アフィニティーをきめ細かに制御することを前提としています。

NUMA ベースのメモリー管理 API によりプログラマーが利用できる機能として、メモリーページの移動があります。一般に、1 つのノードから別のノードへメモリーページを移動するのは、大きなコストがかかる処理であり、できるだけ避けたほうが良いでしょう。実行時間が長くメモリーを大量に消費するアプリケーションは、メモリーページを移行して NUMA に適した構成を再構築する価値があるかもしれません。例えば、長時間実行されるアプリケーションが多くのスレッドを終了させ、別のノードに常駐する新しいスレッドを生成する場合を考えてみます。データを使用する新しいスレッドに対してデータがローカルではないため、アクセス要求の大半は最適ではありません。この場合、アプリケーションにおけるスレッドの存続期間とデータ要件を考慮し、明示的なスレッドの移動を行うべきかどうか判断することができます。

利用ガイドライン

NUMA アーキテクチャーのパフォーマンスの利点が得られるかどうか判断する上で重要なのは、データの配置です。多くの場合、データはそれを必要とするプロセッサーのローカルメモリーに配置したほうが効率的であり、そうすることで全体的なアクセス時間が短縮されます。各ノードに個別のローカルメモリーを持たせることで、メモリーアクセスにおける共有メモリーバスに関連したスループットの制限と競合問題を回避することができます。メモリーが制約型のシステムでは、理論的には完全な並列化によってメモリーにアクセスすることで、システム上のノード数までパフォーマンスを向上できます。

一方、アクセスするノードのローカルメモリーにないデータが多ければ多いほど、このアーキテクチャーではメモリー・パフォーマンスが低下します。NUMA モデルでは、隣接するノードからデータを取得する場合でも、ローカルメモリーにアクセスする場合よりもかなり多くの時間がかかります。一般に、プロセッサーからの距離が遠ければ遠いほど、メモリーアクセスにかかるコストは高くなります。

関連情報

Drepper, Ulrich 著『What Every Programmer Should Know About Memory』2007 年 11 月

インテル® 64 アーキテクチャーおよび IA-32 アーキテクチャー最適化リファレンス・マニュアル』8.8 節「アフィニティーと共有プラットフォーム・リソースの管理」を参照。2009 年 3 月

Lameter, Christoph 著『Local and Remote Memory: Memory in a Linux/NUMA System』2006 年 6 月

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。